Autotuning Skeleton-Driven Optimizations for Transactional Worklist Applications

Autotuning Skeleton-Driven Optimizations for Transactional Worklist Applications

Luís Fabrício Wanderley Góes, *Member*, *IEEE*, Nikolas Ioannou, *Member*, *IEEE*, Polychronis Xekalakis, *Member*, *IEEE*, Murray Cole, *Senior Member*, *IEEE*, and Marcelo Cintra, *Senior Member*, *IEEE*

Abstract—Skeleton or pattern-based programming allows parallel programs to be expressed as specialized instances of generic communication and computation patterns. In addition to simplifying the programming task, such well structured programs are also amenable to performance optimizations during code generation and also at runtime. In this paper, we present a new skeleton framework that transparently selects and applies performance optimizations in transactional worklist applications. Using a novel hierarchical autotuning mechanism, it dynamically selects the most suitable set of optimizations for each application and adjusts them accordingly. Our experimental results on the STAMP benchmark suite show that our skeleton autotuning framework can achieve performance improvements of up to 88 percent, with an average of 46 percent, over a baseline version for a 16-core system and up to 115 percent, with an average of 56 percent, for a 32-core system. These performance improvements match or even exceed those obtained by a static exhaustive search of the optimization space.

Index Terms—Concurrent programming, transactional memory, parallel patterns and application-transparent adaptation

1 Introduction

Leading processor manufacturers have recently shifted toward the multicore design paradigm [1], [2]. As devices continue to scale we can expect future systems to be comprised of an even larger number of cores. Unfortunately, this means that to sustain performance improvements the programmers/compilers now have to exploit the available cores as much as possible through coarse-grain parallelism. Although parallel programming is not a new concept, the vast majority of programmers still find it a hard and error-prone process, especially when based on low-level programming approaches, such as threads with locks [3].

One alternative to simplify the development of parallel applications is to employ parallel algorithmic skeletons or patterns [4], [5], [6]. Skeleton-based programming stems from the observation that many parallel algorithms fit into generic communication and computation patterns, such as pipeline, worklist and MapReduce [7]. The communication and computation pattern can be encapsulated in a common infrastructure, leaving the programmer with only the implementation of the particular operations required to solve the problem at hand. Thus, this programming approach eliminates some of the major challenges of parallel programming, namely thread communication, scheduling and orchestration.

Transactional Memory (TM) [8] is another alternative parallel programming model. From a different perspective, it simplifies parallel programming by removing the burden of correctly synchronizing threads on data races. This model allows programmers to write parallel code as transactions, which are then guaranteed by the runtime system to execute atomically and in isolation regardless of eventual data races. Although removing the burden of correctly synchronizing parallel applications is an important simplification, the programmer is still left with the tasks of thread scheduling and orchestration. These tasks can be naturally handled by a skeleton framework.

Another opportunity provided by skeletons in addition to the simplification of programming is the enabling of performance optimizations. The skeleton framework can exploit pattern, application and/or system information to perform optimizations such as communication contention management and data prefetching. Moreover, such optimizations can be performed transparently, that is, without requiring any additional programming effort from the application programmer. Nevertheless, the decision of which optimizations should be enabled and how to adjust them for a given application is still a daunting task. This issue is commonly tackled by the use of online autotuning mechanisms [9], [10], [11].

In this paper, we combine the worklist parallel skeleton with software transactional memory into a single framework which enables performance optimizations to be transparently selected and applied according to the application behavior. It is based on a novel hierarchical autotuning mechanism that dynamically selects the best performing set of optimizations for each application. This

- *L.F.W. Góes is with the Pontificia Universidade Catolica de Minas Gerais, Rua Salinas 2216, Santa Tereza, CEP: 31015-190 Belo Horizonte, Minas Gerais, Brazil. E-mail: lfwgoes.inf@ed.ac.uk.*
- *N. Ioannou is with the IBM Zurich Research Lab, Säumerstrasse 4, 8803 Rüschlikon, Switzerland. E-mail: nikolas.ioannou@ed.ac.uk.*
- *P. Xekalakis is with the Intel Barcelona Research Center, Intel Labs Barcelona, C/Jordi Girona 29, Nexus II, 08034 Barcelona, Spain. E-mail: polychronis.xekalakis@intel.com.*
- *M. Cole and M. Cintra are with the University of Edinburgh, Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom. E-mail: mic@inf.ed.ac.uk, mc@staffmail.ed.ac.uk.*

Manuscript received 13 Feb. 2012; revised 12 Apr. 2012; accepted 17 Apr. 2012; published online 2 May 2012.
Recommended for acceptance by M.E. Acacio.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-02-0096.
Digital Object Identifier no. 10.1109/TPDS.2012.140.

mechanism incrementally enables and tunes optimizations following their performance impact and abstraction level order. Additionally, it automatically tunes the thread concurrency level.

In order to evaluate our Open Skeleton (*OpenSkel*) autotuning framework, we ported five STAMP benchmarks to conform to its API. Our results on the STAMP benchmarks show that OpenSkel can achieve performance improvements of up to 88 percent, with an average of 46 percent, over a baseline version for a 16-core system and up to 115 percent, with an average of 56 percent, for a 32-core system.

This paper makes the following contributions:

- We propose a skeleton framework based on a novel, dynamic autotuning mechanism that successfully achieves performance improvements that are close to, and sometimes surpass a static oracle.
- We adapt and enable skeleton-driven performance optimizations to be applied transparently and analyze their combined performance impact on the standard, well known, TM benchmark STAMP.
- We combine skeleton-based and transactional programming into a new framework, inheriting the programming and performance benefits of both models.

The rest of this paper is organized as follows: Section 2 describes how skeletons can improve the performance of TM applications. Section 3 describes the proposed skeleton-based framework. Section 4 presents the performance optimizations. Section 5 presents our proposed autotuning mechanism. Section 6 outlines our experimental methodology while Section 7 presents results. Finally, Section 8 discusses related work and Section 9 concludes the paper and points out future work.

2 Background

2.1 Skeletal Parallel Programming

Skeletal programming [5] is a pattern-based approach which proposes that parallel programming complexity should be addressed by extending the programming model with a small number of architecture independent constructs, known as *algorithmic skeletons*. Each skeleton specification captures the behavior of a commonly occurring pattern [12] of computation and interaction, while packaging and hiding the details of its concrete implementation. This both simplifies programming, by encouraging application and combination of the appropriate skeletons, and enables optimizations, by virtue of the *macro* knowledge of application structure that is provided.

Essentially, the skeleton "knows what will happen next" and can use this knowledge to choose and adjust implementation details. For instance, skeleton implementations may be able to place threads that communicate frequently on cores that share some level of cache memory, to prefetch data for the next step of a thread computation, and so on.

A key benefit of skeletons is that the optimizations can be applied *transparently* and *architecture-sensitively*, without user intervention to any application for which the programmer has used the corresponding skeleton. Skeletons form a key component of the Berkeley ParLab software strategy [4], where they are known as "program frameworks," are present in Intel's Threading Building Blocks (TBB) library software [13] in the form of the pipeline and scan operations, and are exemplified by Google's MapReduce paradigm [7].

```
   input: Work-Units u, Worklist W,
          Threads t ∈ T
 1 begin
 2 |   Add seed work-units u_i into W
 3 |   foreach t_i ∈ T do
 4 |   |   while W ≠ ∅ do
 5 |   |   |   Remove a work-unit u_j from W
 6 |   |   |   Process u_j
 7 |   |   |   [Add new work-units u'_j to W]
 8 |   |   end
 9 |   end
10 end
```

Fig. 1. Generic behavior of the worklist skeleton.

2.2 Programming with Transactional Memory

For at least the short and medium term, many-core processors will present shared address space programming interfaces. If a conventional approach is followed, these will present complex, weakened memory models, synchronization built around locks and condition variables. In contrast, the TM model [8], [14] offers both conceptually simplified programming and potential for competitive, or even improved, performance against traditional approaches. In essence, TM requires the programmer to express all synchronization through transactions: blocks of code that must *appear* to execute atomically. The core aim of a TM implementation is to allow as many transactions as possible to proceed concurrently, in potential conflict with the atomic semantics, backtracking (or "aborting") on one or more transactions only when the memory accesses *actually* conflict.

2.3 How Skeletons Can Improve the Performance of Transactional Applications

In order to apply the skeletal programming methodology within the context of TM, we must answer three questions: *What are the relevant skeletons?*, *Which optimization opportunities can be exploited by these skeletons?* and *How can these skeletons be automatically tuned?* The focus of this paper is a TM oriented *worklist* skeleton, for which we have investigated a number of performance optimizations and proposed a novel autotuning mechanism to select and adjust them. The skeleton was derived from a study of TM applications from the STAMP benchmark suite.

A skeleton for transactional applications. Applications that exhibit the *worklist* pattern are characterized by the existence of a single key operation: process an item of work known as a *work unit* from a dynamically managed collection of work unit instances, the *worklist*.

The algorithm in Fig. 1 sketches the generic behavior of worklist algorithms. The worklist is seeded with an initial collection of work units. The worker threads then iterate, grabbing and executing different work units potentially in parallel until the worklist is empty. It is important to note that there is no guarantee about the order in which work

units are executed. As a side effect of work unit execution, a worker may add new work units to the worklist.

Typically, work units access and update common data and require mutual exclusion mechanisms to avoid conflicts and ensure correct behavior. Applications in areas such as routing, computer graphics, and networking [15], [16] are fertile territory for TM programming models, because the number of conflicts which might occur is often much higher than that which actually does occur in typical runs. Our *transactional worklist skeleton* ensures correctness by executing all concurrent computation of work units protected by transactional memory barriers. In Section 3, we present the implementation of this skeleton.

Performance optimization opportunities. The proposed transactional worklist skeleton provides many performance optimization opportunities. These opportunities derive from pattern and Software Transactional Memory (STM) information. First, the worklist pattern carries the important semantics that there is no required ordering on execution of available work units. This allows the implementation to radically alter *the mechanisms by which the worklist is stored and accessed*.

Second, we observe that the worklist pattern does not specify lockstep progression by workers through iterations. This means that an execution in which some worker commits the effects of several work units in sequence, without interleaving with other workers is valid. This gives us freedom to experiment with the *granularity of our transactions*.

Another opportunity stems from the fact that the proposed skeleton deals with transactional applications. Centralized worklists may cause aborts by leading threads to focus activity within small regions of the application data space. These are often unnecessary, since many other *potential work units are available* which were better distributed across the data space can be executed. Transactional applications that present high abort ratio may experience inherent scalability constraints, leading to poor returns for the use of additional cores. This observation creates an opportunity to exploit the *assignment of additional cores to do some other useful computation* that would help to boost the execution of the application as a whole.

These opportunities are exploited in Section 4.

Autotuning skeletons. As mentioned earlier, skeletons have full control of the communication behavior of an application. In particular, for the worklist transactional skeleton, this is done by communicating directly with the underlying TM system and adjusting the worklist structure accordingly. More specifically, the skeleton collects runtime performance measurements such as work unit aborts and commits, the number of stalls to access the worklist and work units throughput. This information, coupled with the current parameters with which the measurements are collected, are then used to drive decisions on how to adjust the behavior so that performance improvements are attained. Adjusting the behavior is done by enabling or disabling performance or even by fine-tuning internal parameters for specific optimizations. In Section 5, we propose a novel online dynamic autotuning mechanism that allows skeletons to automatically adjust to the application behavior.

```
01: int main(void *args)
02: {
03:   file f = open(args[1]);
04:   oskel_wl_shared_t global = malloc();
05:   global -> data = malloc();
06:
07:   oskel_wl_t* oskelPtr =
08:               oskel_wl_alloc(&oskel_wl_initWorker,
09:               &oskel_wl_processWorkUnit,
10:               &oskel_wl_update,
11:               &oskel_wl_destroyWorker);
12:
13:   while (!feof(f))
14:     oskel_wl_addWorkUnit(oskelPtr,read(f));
15:
16:   oskel_wl_run(oskelPtr,global);
17:   oskel_wl_free(oskelPtr);
18: }
```

Fig. 2. The main function of a typical transactional worklist application on OpenSkel.

3 The OpenSkel System

This paper introduces OpenSkel, a C runtime system library that enables the use of the *transactional worklist skeleton*. It provides an API to handle transactional worklists and implements skeleton-driven performance optimizations with autotuning. OpenSkel exploits existing word-based STM systems to deal with transactions. As shown in Fig. 2, the programmer is provided with three basic primitives so as to allocate, run, and free a worklist. Additionally, the API provides a function, namely *oskel_wl_addWorkUnit()*, with which the programmer can dynamically add work units to the worklist.

The OpenSkel API allows the programmer to mainly focus on the kernel implementation to process a work unit in the *oskel_wl_processWorkUnit(void* workUnit, oskel_wl_private_t* p, oskel_wl_shared_t* s)* user function. This function requires as input shared global and private local variables that are declared in the *oskel_wl_shared_t* and *oskel_wl_private_t* data structures, respectively, which in turn are initialized and terminated in the *oskel_wl_initWorker(oskel_wl_private_t* p, oskel_wl_shared_t* s)* and *oskel_wl_destroyWorker(oskel_wl_private_t* p, oskel_wl_shared_t* s)* user functions. Additionally, it provides the *oskel_wl_update(oskel_wl_private_t* p, oskel_wl_shared_t* s)* function that allows the programmer to implement any kind of operation to update the global data when a worker thread is just about to finish.

Once the worklist is loaded with work units, the OpenSkel runtime system takes care of the application execution through the *oskel_wl_run()* call. This function starts all worker threads and waits in a barrier. Fig. 3 shows OpenSkel's internal implementation of each worker thread. Each one coordinates the execution of the aforementioned user functions. After initialization, each worker thread grabs work units with *oskel_wl_getWorkUnit()* and calls the *oskel_wl_ processWorkUnit()* function until the

```
01: void oskel_wl_worker(oskel_wl_t* oskelPtr,
02:                      oskel_wl_shared_t* global)
03: {
04:   void* workUnit;
05:   int tid = getThreadId();
06:   oskel_wl_private_t* local = oskelPtr->locals[tid];
07:   oskel_wl_initWorker(local,global);
08:
09:   do {
10:     atomic {
11:       if((workUnit = oskel_wl_getWorkUnit()))
12:         oskel_wl_processWorkUnit(workUnit,
13:                                  local,global);
14:     }
15:   } while(workUnit);
16:
17:   atomic {
18:     oskel_wl_update(local,global);
19:   }
20:
21:   oskel_wl_destroyWorker(local,global);
22: }
```

Fig. 3. The OpenSkel internal worker pseudocode.

worklist is empty. Although the *oskel_wl_getWorkUnit()* is within a transaction, its variables are not protected by transactional barriers. Instead, this function internally uses locks to access OpenSkel's worklist and internal state. This is essential to decouple the worklist management from the transactional memory system, avoiding extra transaction conflicts and contention.

The *oskel_wl_processWorkUnit()* procedure is executed within transactional barriers placed by the skeleton library. This function is then translated to transactional code at compile time by any existing TM compiler such as Dresden TM [10]. This process is transparent and completely relieves the application programmer of the burden of having to handle transactions explicitly.

4 Skeleton-Driven Optimizations

A skeleton-driven approach makes available useful information about the pattern, application, and system at compile time and runtime. This information allows OpenSkel to provide a set of performance optimizations. In the rest of this section, we present the implementation details of several skeleton-driven performance optimizations to the transactional worklist skeleton. These optimizations have been proposed and used in other contexts [13], [17], [18], [19]. The novelty of our approach is to apply them *automatically* in a transactional skeleton framework driven by an autotuning mechanism that dynamically selects and adjusts them without user intervention, as we describe in Section 5. We start by describing the baseline implementation of our skeleton.

Baseline. There are two important implementation issues to be addressed in the baseline version: 1) the data structure with which to implement the worklist and 2) how to access it.

The worklist data structure can be implemented in many ways. Our priority was to find a simple, efficient, and flexible enough structure to hold different optimizations. Based on that, we have chosen to implement the OpenSkel worklist as a stack under a work sharing scheme, as shown in Fig. 4a. First, work sharing provides a centralized worklist and promotes load balancing. Second, a stack is a fast structure to insert and remove elements since there are no search operations for elements. In our particular case, because OpenSkel does not execute any search operations in the worklist, there is no need for a more complex structure. Additionally, if consecutive elements in a stack are memory correlated, the stack may improve data locality because workers would work within the same memory region. However, this very property is a double-edged sword. Poor exploitation of this behavior could lead to high contention as workers will start competing for the same data.

Access to the worklist can be protected by a transaction or a lock. Since an access to the worklist is short and transactional concurrent accesses to the worklist usually lead to a conflict, we chose to use locks. OpenSkel's baseline version thus uses compare-and-swap-based locks with busy waiting instead of transactions to access the worklist. We expect that under high abort ratio, this synchronization strategy will lower contention for the worklist.

Stealing work units. The first optimization is *"work stealing"* (WS), employed in many systems such as Intel TBB [13] and Cilk [20]. It tackles the contention to access the worklist which occurs with increasing number of work units and worker threads. It exploits the knowledge that work units can be executed in any order and by any core in the system.

We implemented a set of private worklists, with the initial number of work units being split among workers in a round-robin fashion. As shown in Fig. 4b, each worker has its own privatized worklist in which it inserts and removes work units. When a local worklist runs out of work units it steals work units from another worker. This stealing policy is implemented using busy-waiting locks that synchronize the victim and the thief workers, selects a victim in a random fashion, copies half of the work units to the thief worker, removes them from the victim worker and frees both worker threads. This optimization tries to reduce the contention to the worklist and also alleviates aborts.

Coalescing work units. The second optimization exploits the fact that two work units executed in a single transaction are as semantically correct as if executed in separate transactions. We call this optimization *"work coalescing"* (WC) based on the technique proposed in [18]. It executes two work units that would be executed in different transactions, within the same transaction. Fig. 4c exemplifies how this optimization works. The main benefit expected from coalescing work units is to reduce the contention to the worklist and improve cache affinity if consecutive work units are memory correlated. As worker threads grab two work units at once, they take more time to access the worklist again. This reduces the contention over the worklist. As a side effect, these longer transactions may increase the number of conflicts.

Swapping work units. *"Swap retry"* (SR) is our third implemented optimization. This is based on the *steal-on-abort* technique proposed in [17]. Usually, when a transaction

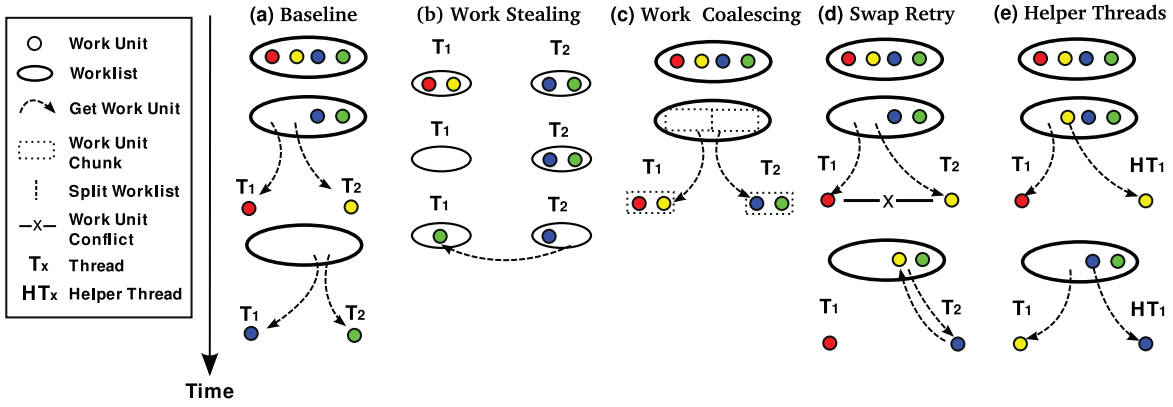


Fig. 4. Skeleton-driven performance optimizations and the baseline version.

aborts, a STM reexecutes the transaction hoping that the conflict will not reoccur. The other common available alternatives to the STM are to assign a higher priority to the transaction or wait for a time interval before reexecuting it. Nevertheless, a STM does not have an alternative to try to execute a different transaction, unless explicitly implemented by the programmer through a retry function [16].

In a transactional worklist skeleton, it is possible to try a different work unit since the skeleton has full control over the worklist and work units can be executed in any order. This optimization takes advantage of this high-level information. As shown in Fig. 4d, when a transaction aborts, this optimization swaps the current work unit with another one before it reexecutes. The stack-based worklist was extended to support the swap operation. In this operation, OpenSkel randomly selects one work unit based on the size of the stack, and swaps the top or current work unit of the stack with this selected work unit.

In this way, transactions that keep aborting can be postponed and executed later. *Swap retry* is employed to reduce aborts. However, one side effect is to eagerly swap work units and discard the data prefetching made by aborted executions of the same transaction. To alleviate this problem, we introduce a parameter for the number of retries. The swapping is only actually done after a transaction reaches a particular number of retries.

Employing helper threads (HTs). Another optimization that we employ is to perform data prefetching using automatically created *"helper threads."* They are auxiliary threads that run concurrently with a main thread. Their purpose is not to directly contribute to the actual program computation, which is still performed in full by the main thread, but to facilitate the execution of the main thread indirectly. Typically, modern multicores have at least one shared level of cache among the cores, so that HTs may try to bring data that will be required by the main thread into this shared cache ahead of time. Helper threads have previously been developed in software [19] and hardware [21].

TM applications have a number of characteristics that render the use of HTs appealing. First of all, some transactional applications do not scale up to a large number of cores because the number of aborts and restarts increases. If more cores are available, we can use them to run HTs instead of more TM threads and thus improve the performance of our applications. Another characteristic of STM applications is the high overhead and cache miss ratio of transactional loads and stores. This suggests that HT can more easily stay ahead of the main thread while effectively prefetching for it.

Unfortunately, a STM does not have the required information to implement HT on its own. The worklist skeleton, on the other hand, provides two key pieces of information to make HTs feasible: when to start a HT and which data to prefetch. As we observe in Fig. 4e, every time a worker thread starts computing a work unit, a HT should start computing the next work unit assigned to the worker in the worklist.

Helper thread code is generated and instrumented in the same way compilers like Dresden TM compiler would do for STM systems. However, instead of function calls to the STM system, they use modified functions for reading and writing shared variables. In contrast to normal TM threads, every time a HT has to access a shared global variable it has to use special functions to redirect accesses to the internal metadata structures managed by the OpenSkel runtime system.

As HTs do not change the state of the application, each write to a global variable is done in its local entry in a hash table rather than in the actual memory location. If the same variable is read after being written, the value will be extracted from the hash table instead of the actual memory location. This enables HTs to hopefully follow the correct path of control and prefetch the correct data. However, if a transaction modifies shared data, the HT may go down the wrong path, possibly prefetching wrong data or even worse, raising exceptions (e.g., a segmentation fault) that could crash the whole application. OpenSkel HT thus implements a transparent mechanism to deal with exceptions. If an exception is raised, the OpenSkel library aborts the helper thread and restarts it in a barrier waiting for the next work unit.

To ensure that a HT is prefetching data to the right place, the OpenSkel checks whether there are cores that share any level of cache memory. If so, it schedules each pair of worker thread and helper thread to cores that share the same level of cache. To reduce cache pollution due to inefficient prefetching, we employ a lifespan parameter (i.e., number of words prefetched per work unit) and limit the hash table size.

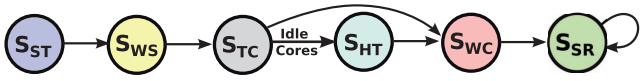


Fig. 5. Flow diagram of our autotuning mechanism. Each state accounts for the tuning of a single optimization.

5 Autotuning Optimizations

Although enabling many skeleton-driven optimizations transparently in a single framework is an important step, the application programmer would still be left with the daunting task of choosing the most profitable set of optimizations. If the programmer chooses the wrong set of optimizations, he could potentially degrade the peformance of the applications. In order to tackle this issue, we propose a novel hierarchical autotuning mechanism. In addition to enabling the most efficient set of optimizations, it also adjusts each optimizations internal parameters. Additionally, this mechanism also tunes the number of concurrent threads automatically.

Each pattern-oriented optimization commonly tackles different performance bottlenecks. This creates an opportunity to combine more than one optimization to improve performance even further. Additionally, the presented pattern-oriented optimizations are orthogonal. As a result, the OpenSkel framework can enable multiple optimizations simultaneously. However, the dynamic selection of the best performing set of optimizations is still dependent on the order in which these optimizations are activated. This stems from the fact that enabling a specific optimization can influence the performance of a subsequent optimization. For instance, if the number of concurrent threads are adjusted before the worklist stealing optimization, it may lead to poor performance since an application may scale to a higher number of cores after enabling the *work stealing* optimization. Additionally, there are some particular constraints that have to be taken into account in the selection of an autotuning strategy. First, the optimization space is significantly large since some optimizations also have internal parameters to be tuned (e.g., *swap retry*). Second, transactional applications usually exhibit short execution times [22]. The autotuning mechanism thus has limited time to converge to a set of tuned optimizations. Finally, as mentioned before, some optimizations have more performance impact than others, meaning that the order in which optimizations are enabled is important.

In this paper, we thus propose an online autotuning mechanism that *boots* optimizations in a specific order determined by their performance impact on scalability. On each iteration of the mechanism, it dynamically improves the selected current set of optimizations based on dynamic information about the application behavior. This one-factor-at-a-time method simplifies the design and improves the interpretability of the autotuning mechanism. Additionally, this method allows the autotuning mechanism to be easily extended with new optimizations. Backed by the results presented in Section 7, it was possible to evaluate the performance impact of each optimization individually and determine a predominance order between these optimizations. Fig. 5 illustrates the autotuning mechanism in the form of a state diagram that takes into account this predominance order.

The initial state is the start state (S_{ST}), which initializes the autotuning mechanism and triggers a collection of statistics such as number of aborts, stalls, and commits. Then the autotuning mechanism moves onto the worklist sharing state (S_{WS}). In this state, it decides to enable or not the *Work Stealing* (WS) optimization.

The proposed autotuning mechanism also tunes the number of concurrent threads. The aforementioned results showed that the most profitable number of threads changes if the WS optimization is activated. This means that the enabling of the WS optimization affects the application scalability potentially allowing more threads to run concurrently without degrading performance. For this reason, S_{WS} precedes the thread counting state (S_{TC}). Once S_{WS} is finished, the autotuning mechanism then adjusts the number of concurrent threads in S_{TC}.

The other optimizations do not affect scalability, allowing them to be enabled in a more flexible order. It was decided then that the next natural step was to check if there are any cores still available in order to enable and adjust *helper threads* in the helper thread state (S_{HT}). In particular, the autotuning mechanism fine-tunes HTs by adjusting their lifespan parameter. Fig. 9 in Section 7 shows that HTs are not sensitive to application phases. As a result, it would be too costly and inefficient to continuously tune HT. Additionally, the buffer size internal parameter does not make a significant enough impact to justify its inclusion in the tuning process. For these reasons, the autotuning mechanism adjusts only the lifespan of the HTs during the S_{HT} state and uses a fixed buffer size.

If there are no idle cores left, then the mechanism moves directly to the work coalescing state (S_{WC}). The *work coalescing* optimization is only enabled in a very specific scenario since the WS optimization has already tackled the same performance bottleneck (i.e., contention to access the worklist) for most scenarios in the S_{WS} state. Finally, the mechanism enables and keeps tuning the *swap retry* optimization in the *swap retry* state (S_{SR}), until the program ends. It is assumed that swapping work units is always beneficial since contention to access the worklist has been already alleviated by the preceding optimizations.

All the pattern-oriented optimizations could be continuously tuned throughout the application execution. It is known that some applications have multiple transactional phases [11], [22]. In this case, an application can benefit from a continuous tuning mechanism. Among the available optimizations in OpenSkel, SR requires a small runtime overhead and is more sensitive to the application execution phases (i.e., to the variation in the STM contention measured by the transaction abort ratio). Some of the optimizations that we tune, however, are not very dependent on the transactional behavior. This is true for the thread count and helper thread optimizations. Moreover, the tradeoff between the overhead of tuning these optimizations versus the gains of dynamically adapting to the phases favors the former. We thus chose a hill-climbing approach in tuning the thread count and helper thread optimizations, while we dynamically tune the SR one.

```
   input : State S, Number of Work-Units I,
           Number of Aborts A, Number of Stalls B,
           Number of Commits C, Lifespan L,
           Number of Threads N, Number of Cores P
   output: State S, Number of Threads N,
           Lifespan L, Number of Retries R
1  begin
2      if C mod sqrt(I)/P = 0 then
3          if S = S_ST then
4              S ← S_WS
5              N ← P
6              L ← 1
7          else if S = S_WS then
8              if B/C > α then
9                  optWorkStealing ← true
10                 S ← S_TC
11             end
12         else if S = S_TC then
13             autoTuningTC()
14         else if S = S_HT then
15             optHelperThreads ← true
16             autoTuningHT()
17         else if S = S_WC then
18             if A/C < β/10 and B/C > α then
19                 optWorkCoalescing ← true
20             S ← S_SR
21         else if S = S_SR then
22             optSwapRetry ← true
23             R ← 2^⌊(A/(A+C))×10
24         end
25     end
26 end
```

Fig. 6. The main algorithm of the proposed autotuning mechanism.

```
   input : State S, Counter R, Number of Aborts A,
           Number of Commits C,
           Number of Threads N, Number of Cores P
   output: State S, Counter R, Number of Threads N
1  begin
2      if A/C < β then
3          if R = 0 then
4              R ← R + 1
5          else if R = 1 then
6              if N < P then
7                  S ← S_HT
8              else
9                  S ← S_WC
10             end
11         else
12             R ← 0
13         end
14     else
15         if R = 0 then
16             R ← R − 1
17         else if R = −1 then
18             N ← max(N/2, 1)
19             R ← 0
20         else
21             R ← 0
22         end
23     end
24 end
```

Fig. 7. The algorithm to autotune the thread concurrency level in the autoTuningTC() function.

```
   input : State S, Counter R, Lifespan L,
           Number of Commits C, Epoch E,
           Max Lifespan M
   output: State S, Counter R, Lifespan L
1  begin
2      if C_i/E_i < C_{i−1}/E_{i−1} then
3          if R = 0 then
4              R ← R − 1
5          else if R = −1 then
6              S ← S_WC
7              if L = 1 then
8                  L ← 0
9              end
10         else
11             R ← 0
12         end
13     else
14         if R = 0 then
15             R ← R + 1
16         else if R = 1 then
17             L ← min(L × 10, M)
18             R ← 0
19         else
20             R ← 0
21         end
22     end
23 end
```

Fig. 8. The algorithm to autotune the lifespan of helper threads in the autoTuningHT() function.

The main algorithm of the autotuning mechanism is shown in Fig. 6. It implements the state diagram depicted in Fig. 5. First, the application starts with the default baseline version with an optimistic number of threads, that is, the maximum number of available cores. This avoids the case where the application loses any available parallelism in its first iterations. However, as a side effect, this approach increases the contention and may slowdown applications with low parallelism. Additionally, the state variable S is initially set to the initial state S_{ST}. Then, on each iteration, the main worker thread (i.e., thread $\mathrm{id} = 0$) calls the main algorithm before grabbing a new work unit. The frequency at which the tuning process actually happens is proportional to the number of initial work units I and cores P, as depicted on line 2 in Fig. 6. On every $\sqrt{I}/P$ committed work units, defined as an *epoch*, the autotuning mechanism reevaluates its current state. This usually results in a change of state and/or the enabling of an optimization. The latter is represented by the assignment of *true* to the corresponding optimization variable. For instance, line 9 shows when the WS optimization is activated. In particular, some states take several epochs to switch to the next state. This is the case for the S_{TC} and S_{HT} states in which the *autoTuningTC()* and *autoTuningHT()* functions, on lines 13 and 16 in Fig. 6, implement, respectively, the hill-climbing strategies to tune the number of threads and the lifespan of helper threads. These functions are presented in detail in Figs. 7 and 8. Finally, the α and β thresholds, on lines 8 and 18 in Fig. 6, are determined at design time by a sensitivity analysis that is discussed in Section 7. The rest of this section describes the implementation details to tune each of the pattern-oriented optimizations.

After the initial S_{ST} state, the autotuning mechanism moves to the S_{WS} state. In this state, it evaluates if there is high contention to access the worklist as a condition to enable the WS optimization. In order to do this, it checks if the ratio between the number of *stalls* to access the worklist and the number of *committed* work units is above a threshold α as shown on line 8 in Fig. 6. Each time a thread access the worklist and has to wait in a lock, this is counted as a stall. Low stall ratio means that the worklist is not under contention. In this scenario, the work sharing should be maintained since it provides optimal load balancing.

The next step is to adjust the number of threads in the S_{TC} state. The proposed mechanism uses a hill-climbing heuristic implemented in the *autoTuningTC()* function as depicted in Fig. 7. It is based on the ratio between work units *aborts* and *commits*. If this ratio is below a threshold β this means that the actual number of threads exploits parallelism efficiently. However, this has to be confirmed in one more epoch through a *counter* R before fixing the number of threads. In particular, this counter R can assume only three values +1, 0, and -1. This double checking avoids making a wrong decision based on a biased interval. The same is valid when the ratio is above β. The algorithm waits for a consecutive confirmation before halving the number of threads. This process ends when consecutive epochs present a ratio below β. In Section 7, a sensitivity analysis is performed in order to choose and understand the performance impact of these thresholds. In particular, it shows that there is small performance variation on most of the investigated space. It means that even when the mechanism uses nonoptimal threshold values, it still makes the right choices.

Next, the algorithm switches to the S_{HT} state if there are idle cores or goes straight to the S_{WC} state. The autotuning strategy to determine the lifespan of *helper threads* follows a similar approach to the S_{TC} state. As shown in Fig. 8, it also uses a hill-climbing strategy but in the opposite direction. In contrast, it starts with a pessimistic lifespan equal to one and moves toward a maximum lifespan. Basically, if the current throughput with HT enabled is higher than without them, it multiplies the lifespan by a factor of 10. Once the lifespan is determined, the algorithm switches to the S_{WC} state.

The S_{WC} state is implemented on line 17 in Fig. 6. Since the WC optimization also tackles the contention problem in the worklist, it is only enabled if the number of stalls is high and the ratio between aborts and commits is very low, a tenth of β. Then, it switches to the S_{SR} state.

Finally, in the last state, the SR optimization is enabled and adjusted continuously. It uses an exponential function based on the abort ratio to adjust the number of retries before a swap as described on line 23 in Fig. 6. The intuition behind it is that as the abort ratio increases, work unit swaps become expensive and inefficient. This stems from the fact that under high abort ratio, when a work unit is swapped it will end up aborting anyway and the thread may also lose the natural prefetching of the previous execution. On the other hand, it is also assumed that aggressive SR (i.e., swap on every abort) is not beneficial since in the next execution the work unit can execute and commit faster using the prefetched data. Thus, the number of retries is limited to a minimum of one retry before swapping.

TABLE 1
Summary of STAMP Application Runtime Characteristics on TinySTM for the 32-Core NUMA Machine

Application	Scalable up to # Cores	Transaction Abort Ratio	L3 Cache Miss Ratio	Transaction Exec. Time
Intruder	4	high	medium	short
Kmeans	8	high	high	medium
Labyrinth	32	medium	low	long
Vacation	16	low	low	short
Yada	16	high	medium	medium

6 Evaluation Methodology

6.1 Experimental Setup

We conducted tests both on a 16-core Uniform Memory Access (UMA) machine with Intel Xeon E7320 CPU clocked at 2.13 GHz, 64 GB of RAM, 16 MB of L2 cache (2 MB shared per core pair), running Linux kernel version 2.6.18, and on a 32-core Nonuniform Memory Access (NUMA) machine with Intel Xeon x7560 CPU clocked at 2.27 GHz, 64 GB of RAM, four nodes each with a 24 MB L3 cache memory shared by eight cores, running Linux kernel version 2.6.32-5. All code was compiled using GCC version 4.1.2 (16-core) and 4.4.5 (32-core) with the -O3 option enabled.

We selected TinySTM [10] as the STM platform. It can be configured with several locking and contention management strategies. We configured TinySTM with encounter-time locking, write-back memory update and a commit suicide contention strategy.

6.2 Analyzing the STAMP Benchmark Suite

To investigate performance tradeoffs of the optimizations for transactional worklists under the OpenSkel system, we selected five applications from the STAMP benchmark suite [22] that matched the worklist pattern: *Intruder*, *Kmeans*, *Labyrinth*, *Vacation*, and *Yada*. Other applications from STAMP present an irregular behavior with specific characteristics which make them hard to generalize to a single skeleton, such as an arbitrary number of synchronization barriers and very fine-grained transactions. We executed all selected STAMP applications with the recommended input data sets. *Kmeans* and *Vacation* have two input data sets, high and low contention. As *Intruder* and *Yada* only have high contention input data sets, we chose the low contention inputs for *Kmeans* and *Vacation* to cover a wider range of behaviors. We profiled these five applications using TinySTM according to four criteria: scalability, transaction abort ratio, L3 cache miss ratio, and transaction execution time (i.e., average time to execute a transaction).

The results are summarized in Table 1, which demonstrates that our applications span a varied range of points in the implied space, and so provide a sound basis for evaluation.

6.3 Porting to OpenSkel

To port the selected applications, we decomposed single worker functions into the ones required by OpenSkel, grouped global, and local variables into the new declared structures and declared a work unit structure when it was not already there. The code transformation was straightforward for all five applications. We believe that implementing

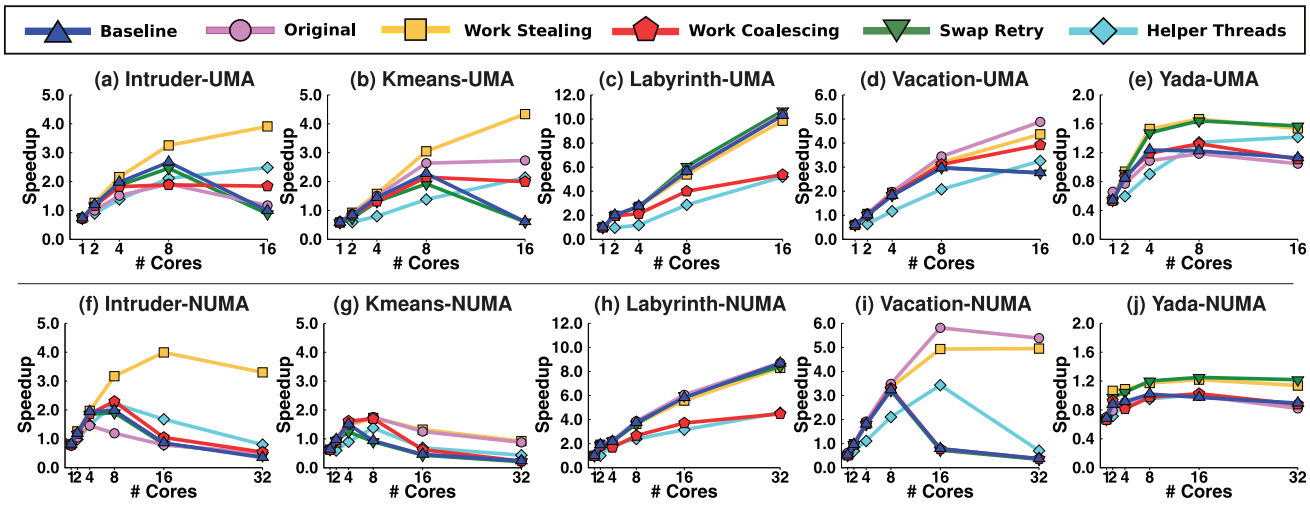


Fig. 9. Speedup on STAMP applications for the skeleton-driven performance optimizations.

an application from scratch under the OpenSkel philosophy would be even more intuitive.

Regarding the transactional code transformation, we decided to use the manually instrumented transactional code available in the STAMP benchmark suite, since [23] showed that the code generated by the OpenTM compiler achieves very similar performance. As helper threads require the same kind of code transformation, we used a script to copy and transform the instrumented transactional code into a helper thread version. It basically substitutes each call to the STM system to a corresponding one in OpenSkel's internal helper thread API.

Some compromises have to be made when existing transactional applications are ported to OpenSkel. These implementation decisions may impact the performance of the OpenSkel version compared to the original application.

First, the worklist data structure in OpenSkel is implemented as a stack. This will influence the ordering and amount of time to access work units. For instance, the original *Yada* and *Intruder* implement the worklist as a heap and as a queue, respectively. Second, as mentioned before, the application has to have work units declared as a structure. *Vacation* and *Kmeans* do not have an explicit worklist in the original benchmark, although they match the worklist pattern. Allocating and handling these work units may introduce some overhead. Particularly in the original *Vacation*, each thread has a fixed number of requests generated randomly in a distributed fashion. The introduction of this new explicit worklist centralizes the generation and access to all work units, which in turn may become a bottleneck.

Transactional applications may split the processing of a work unit into a few phases. Each phase is executed within a fine-grained transaction to reduce the number of aborts. For example, *Yada* and *Intruder* use more than two transactions to process each work unit. On the other hand, to free the programmer from the burden of handling transactions explicitly, each work unit in OpenSkel is processed within a single coarse-grained transaction. However, a single coarse-grained transaction may increase the number of aborts, since each transaction becomes longer. Fortunately, if the majority of aborts are concentrated into a single phase, combining multiple transactions into a single transaction does not become a bottleneck. This is the case for *Intruder* and *Yada*. Since the other applications use only a single transaction to process each work unit then their performance is not affected. Another issue is thread mapping and scheduling. STMs do not manage threads and thus they are left with the operating system default scheduling strategy. The Linux scheduling strategy in the selected platforms and applications tends to map threads initially following the *scatter* mapping strategy. *Scatter* distributes threads across different processors avoiding cache sharing between cores in order to reduce memory contention. However, at runtime the Linux scheduler migrates threads trying to reduce memory accesses and I/O costs. The baseline version employs a static *scatter* mapping strategy in which threads are not allowed to migrate at runtime, guaranteeing a more predictable performance.

In the next section, we show results of the original version for all applications. These results on the STAMP applications show that the baseline version can achieve similar performance, on average, compared to the TM original version on both machines. However, we focus our analysis on the skeleton baseline version since all optimizations are built on top of it.

7 Experimental Results

In this section, we analyze the performance of our skeleton framework. In Fig. 9, the speedup is calculated based on the execution time of the sequential (hence transactionless) version of each application. For all transactional worklist applications, the input work units are shuffled before we start computing them. This is done to avoid benefits from a particular input order. For the *helper threads* optimization half the cores run transactional threads and the other half run helper threads. Due to this, we present results from two to the maximum number of cores for *helper threads*. Nevertheless, we expect them to be profitable only for eight or more cores, depending on when the baseline version stops scaling. All the

results presented (i.e., speedups, performance improvements, etc.) are based on an arithmetic mean of 10 runs.

In the rest of this section, we first analyze the performance benefits of individually applying each optimization. Then, we analyze the performance of our autotuning mechanism compared to the best single optimization and a static oracle extracted from exhaustive space search of all the possible combinations of available optimizations. Finally, we investigate even further the autotuning mechanism behavior by performing a sensitivity analysis on its internal parameters and showing its execution details.

7.1 Analyzing Single Optimizations

In this section, we analyze the performance of each optimization enabled on top of the baseline version.

Work stealing. As we discussed in Section 4, *work stealing* is effective in reducing the contention to access the worklist. According to Fig. 9, it improves the performance of most applications, up to 102 percent for *Intruder*. With the increasing number of cores, applications that execute small to medium transactions are bound to stop scaling due to contention to a centralized worklist. *Work stealing* significantly reduces the contention in the worklist, splitting it between threads. The exception is *Labyrinth*, which executes a small number of long transactions. When the number of threads increases the time to search for a victim with a nonempty worklist also increases. This stems from the fact that *work stealing* selects a victim in a random fashion. Since Labyrinth is left with just a few long transactions in the end of its execution, worker threads may have zero or just one work unit to be stolen. Thus, this search process takes longer to converge, that is, to find a suitable target victim.

For *Kmeans* and *Vacation*, *work stealing* significantly improved their performance over the baseline version. Furthermore, the internal implementation of the OpenSkel worklist as a stack enabled the exploitation of a specific property of *Yada*. Each new bad triangle added to the worklist is correlated with the previous one. Since the *work stealing* optimization has independent stacks for each thread, this leads to two beneficial effects. First, threads stop processing triangles in the same neighborhood, avoiding conflicts. Second, there is a natural prefetching mechanism as a thread will keep working in the same neighborhood. WS was able to reduce the number of aborts of *Yada* and achieve performance improvements up to 35 percent over the baseline version.

Work coalescing. This is designed to be efficient in a low abort ratio scenario. It can reduce contention to the worklist at the cost of increasing aborts. Despite this side effect, it can add up to 32 percent performance on top of the baseline.

As expected, *Labyrinth* had no improvement since combining long transactions within a single transaction leads to high abort ratio. On the other hand, transaction aborts are not an issue for *Vacation*, the abort ratio being less than 20 percent on both STMs and machines. This enabled the *work coalescing* optimization to perform 32 percent better than the baseline (Fig. 9d) due to a reduction in the contention to the worklist.

In the NUMA machine, the contention problem becomes even worse due to the increase in the number of remote accesses to the worklist. In this case, *work coalescing* was able to add up to 16 percent performance for *Intruder* since it presents high contention. This shows that reducing the contention to access the worklist is as important as reducing the number of aborts in a transactional worklist application. Finally, even with an increased abort ratio, *work coalescing* performed 7 percent faster than the baseline for *Yada* in the 16-core machine.

Swap retry. This is an optimization that tackles transaction aborts by swapping conflicting work units to other available ones. According to Figs. 9e and 9j, SR improves the performance of *Yada* by up to 33 percent. *Yada* presents a high abort ratio enabling SR to fit in nicely reducing the abort ratio. Since new inserted work units are spatially correlated in *Yada*, SR also avoids having threads working on the same region.

The *swap retry* optimization adds accesses to the global worklist for each swapping operation. Under high abort ratio, these extra accesses will contribute to increase the contention to the worklist. Thus any performance benefit attained from reducing transaction aborts will be outweighed by the contention overhead. Due to this behavior, SR does not provide any performance improvement to *Kmeans* and *Intruder*.

Vacation has a very low transaction abort ratio, thus *swap retry* is rarely invoked and does not impact on performance. Finally, *Labyrinth* shows a slight 4 percent performance improvement in the 16-core machine as we observe in Fig. 9c.

Helper threads. This optimization can be profitable only when an application stops scaling, leaving idle cores to run helper threads. Even so, we show *helper threads* for all number of cores. In Figs. 9c and 9h, we see that *Labyrinth* scales up to the maximum number of cores. *Labyrinth* thus cannot be improved by HTs in our machines. For the other applications, *helper threads* performed up to 20 percent faster than the baseline version. *Vacation* exhibits substantial cache miss ratio, alleviated with HTs, up to 10 percent.

Yada benefits from the use of *helper threads* and in fact this benefit increases as we increase the number of cores in the UMA machine. As the abort ratio does not increase proportionally to the number of concurrent transactions, the abort ratio per thread is actually reduced. On the NUMA machine, *helper threads* only increase the memory pressure over the shared L3 cache memory thus degrading performance. An exception is Intruder that improves 12 percent over the baseline version. Since it stops scaling with only four cores, adding four more *helper threads* leaves only one main thread and one helper thread per node. That is, each L3 cache is shared by nonconcurrent threads, and so avoids having competing transactions evicting each others cache lines.

7.2 Autotuning and Combined Optimizations

After analyzing the impact of each optimization individually, we now compare the performance of the autotuning mechanism that we presented in Section 5 against the best performing optimization, performed in isolation, and against the best combination of optimizations. Note that the best performing single optimization or combination of optimizations can be different for each workload. The best performing combination of optimizations per workload consists our static oracle.

In Fig. 10, we show the performance improvement of the best single optimization, the autotuning version, and the static oracle over the best *baseline* execution. By best we mean the fastest execution for a specific benchmark, from all possible number of cores (i.e., in some cases more cores result in slowdowns). The static oracle result was obtained through exhaustive investigation of the search space of all combinations of optimizations. Since it is static, that is, optimizations

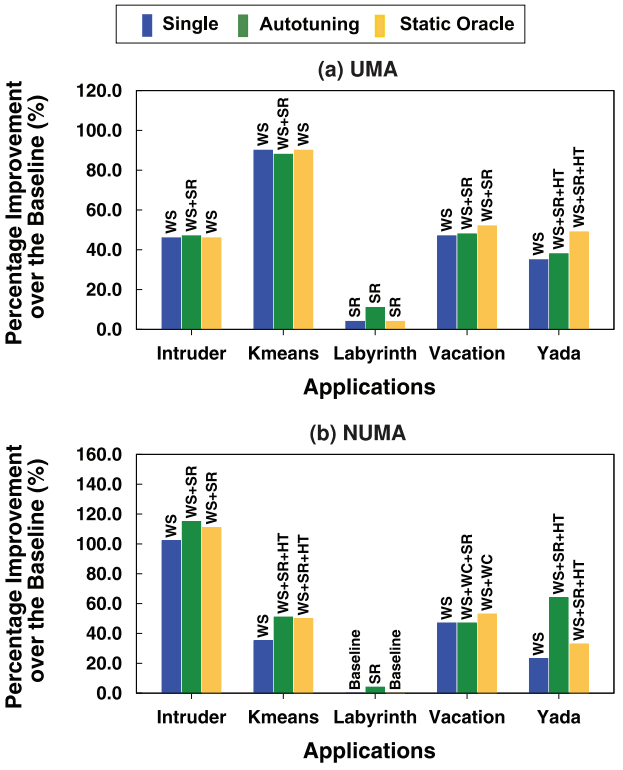


Fig. 10. Performance improvement of the autotuning and static oracle over the best baseline execution.

and their parameters are fixed throughout the application execution, it is possible for the dynamic autotuning mechanism to outperform it. We first discuss the performance benefits of combining optimizations and then compare the static oracle with the autotuning mechanism.

Fig. 10 shows that the combined optimizations for the static oracle deliver significant performance improvements, when compared to single optimizations. As expected, *work stealing* is beneficial for all applications except *Labyrinth*. *Labyrinth* reaches its best performance with the baseline version for the 32-core machine and with *swap retry* for the UMA machine.

As mentioned earlier, under high abort ratio, *swap retry* increases the contention to access the centralized worklist. However, since the *work stealing* optimization privatizes the worklist, this undesired behavior is largely negated when the two optimizations are combined. Since *swap retry* only swaps work units within its local worklist, it was able to deliver performance improvements for many applications.

Work stealing and *work coalescing* tackle the same performance bottleneck, namely contention to access the worklist. Since the *work stealing* benefits outweigh the ones of *work coalescing*, combining them unnecessarily increases the abort ratio in most cases. An exception to this is *Vacation* that has very small transactions and very low abort ratio. Enabling WC on top of WS, reduces contention even further. Finally, combining HTs and WS improved performance even further in *Yada* and *Kmeans*, showing that idle cores can be utilized for prefetching.

7.3 Investigating the Autotuning Behavior

In order to better understand the behavior of the proposed *autotuning mechanism*, we performed a sensitivity analysis for its internal parameters α and β. As mentioned in Section 5,

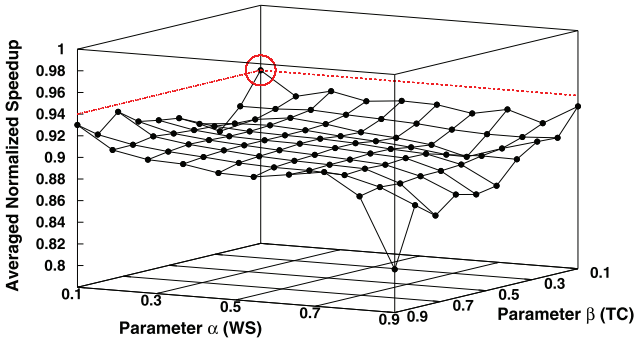


Fig. 11. Sensitivity analysis of the autotuning parameters α and β.

the α threshold is related to the ratio between stalls and commits and it is used in the WS step. The β threshold defines the desired ratio between aborts and commits and is used to choose the number of concurrent threads.

In Fig. 11, we show the average normalized speedup of all the applications with the autotuning enabled running on the 16-core machine. We observe that the best tuple is (0.1, 0.1). This means that even under low contention, the worklist should be split using the WS optimization. Additionally, the number of threads has to be reduced until a low ratio between aborts and commits is reached, so as to achieve the best performance. In contrast, the worse results are achieved with the (0.9, 0.9) tuple. In this case, WS is never used and the number of threads is always the maximum number of cores, which is not profitable for some applications.

Based on this study, we chose the tuple (0.1, 0.1) to drive the proposed mechanism. The same tuple was also used for the 32-core machine, showing that it can be portable across machines. However, we acknowledge that the choice and sensitivity of α and β remains an important topic for further investigation. Fig. 10 shows that the automatic approach can deliver similar, or in some cases even better, performance when compared to the best static combination of optimizations. The proposed mechanism was only 2 percent slower than the best combination on average for the 16-core machine and 15 percent faster for the 32-core one. Note that, however, our autotuning approach performs well—or better than—the best static choice of a single optimization. Moreover, the continuous tuning of *swap retry* proved to have a significant impact for the NUMA machine, improving the performance of *Yada* by 31 percent over the static oracle combination of optimizations.

In Fig. 12, the behavior of the autotuning component is analyzed further for all applications, running on the UMA platform. Each graph shows when each of the states are activated, how many epochs it takes to finish and the values for each of the optimization internal parameters. For instance, Fig. 12c in the S_{SR} state shows that for *Labyrinth* the number of retries remains constant across epochs. Note that it took around 3 percent of *Yada's* total execution time to autotune the application as shown in Fig. 12e. This can be calculated dividing the tuning time (number of epochs until the S_{SR} state) by the total number of epochs. This small tuning time overhead is consistent across all applications, except *Labyrinth* which has long transactions. Despite this fact, it takes a small number of epochs to converge since it is profitable to make use of all cores.

Another interesting result derived from these graphs is that they show how the SR optimization exploits the variation

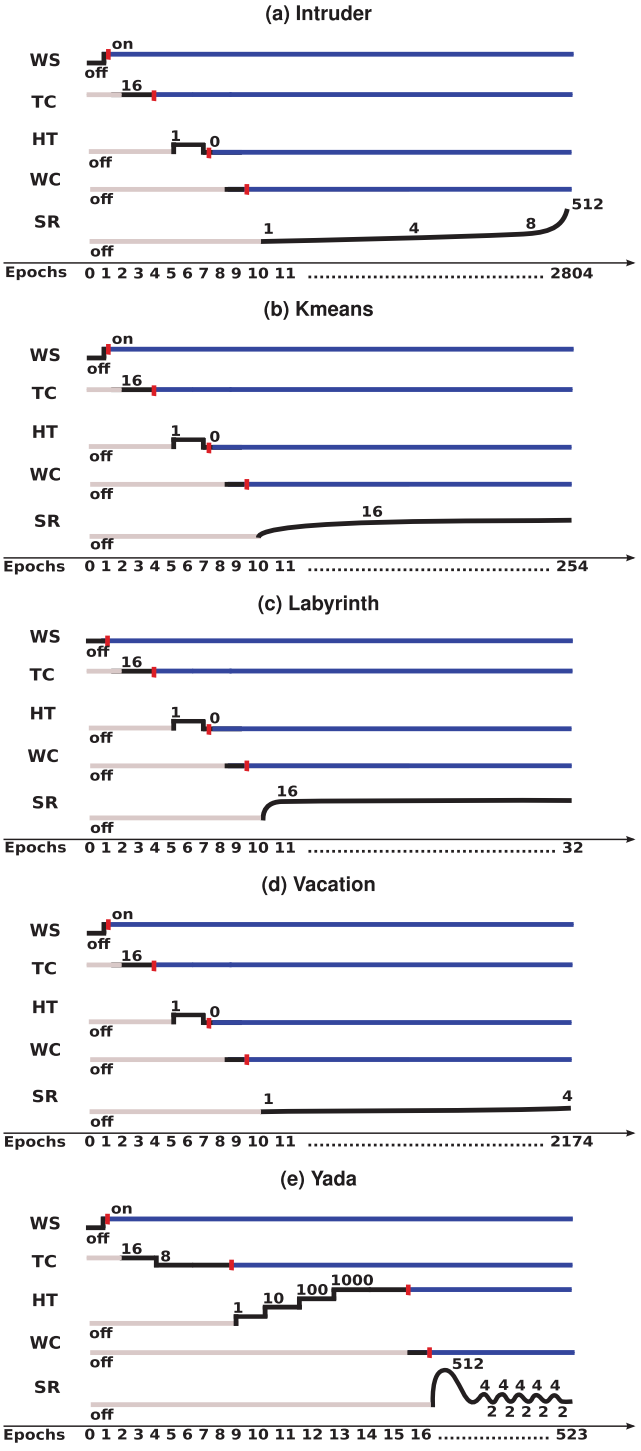


Fig. 12. Dynamic behavior of the autotuning mechanism for each application running on the 16-core platform.

of the abort ratio throughout the application execution. For instance, the number of aborts increases radically by the end of *Intruder*'s execution. Since the number of retries is proportional to the abort ratio, this increase can be observed in the S_{SR} state in Fig. 12a. SR thus almost stops swapping work units since it will be probably worthless as aborts become inevitable.

In Fig. 12e, it is also important to observe that HTs are enabled and the lifespan is increased until 1,000 words per work unit which is an intermediate value. This shows that the autotuning mechanism was able to detect the most profitable lifespan for *Yada* in the UMA platform.

Overall, the proposed autotuning mechanism converges to a set of optimizations that matches the ones of the static oracle. Additionally, it surpasses the static oracle performance benefits when the dynamic tuning of SR is profitable. Finally, it can achieve performance improvements of up to 88 percent, with an average of 46 percent, over a baseline version for the UMA platform and up to 115 percent, with an average of 56 percent for the NUMA platform.

8 Related Work

Autotuning STM systems. In [24], the authors use a history-based heuristic to dynamically select the most efficient STM design considering dimensions such as level of indirection and type of nonblocking semantics. Automatic tuning of STM systems is considered in [10], which proposes a hill-climbing strategy to adjust the number of locks, the number of shifts in a hash function, and the size of the hierarchical array implemented on TinySTM. In contrast, our work focuses on the tuning of higher level optimizations rather than STM internal designs and parameters. Transactional Memory contention managers ([16], [25]) dynamically react to contention and try to minimize it by changing the *conflict resolution* policy, for example, in order to improve performance. Our scheme does take contention into account, but targets a broader range of optimizations. In [11], the authors propose an adaptive mechanism to transaction scheduling on top of a STM contention manager. It uses contention feedback information to adjust the number of available active transactions running in parallel to avoid high contention. In contrast, our autotuning mechanism follows a hill-climbing strategy to select the most suitable number of threads in a few intervals rather than during the whole application execution. In [9], the authors propose an online autotuner called Perpetuum in the operating system level for multicores. It improves the performance of applications as long as they expose their performance-relevant parameters to the operating system. In our approach, this information is implicitly provided by the skeleton framework.

Performance optimizations. Recent work [18] has exploited structure-driven optimizations for irregular applications. In fact, they proposed a technique called *iteration coalescing* that we adapted to *work coalescing* in our skeleton framework. Ansari et al. [17] propose a *"steal-on-abort"* mechanism. When concurrent transactions conflict, the aborted transaction is stolen by its opponent transaction and queued to avoid simultaneous execution. Multiple local queues with job stealing for load balancing are also employed. In contrast, rather than steal a conflicting transaction from another thread, our *swap retry* optimization tries to compute a different work unit from its own stack whenever it aborts. It does not require visible readers and can be applied to any word-based software transactional memory system. Moreover, Ansari et al. focus on a single optimization within a software transactional memory system, where our skeleton-driven approach has a set of transparent optimizations.

One of the applied optimizations exploits the use of helper threads to improve performance. Helper threads have been explored as a means of exploiting otherwise idle hardware resources to improve performance on multicores [21]. A compiler framework to automatically generate software helper threads code for profitable loops in sequential applications has been developed [19]. As in our

system, it prefetches data to shared cache levels. Finally, Nikas et al. [26] manually coded helper threads, within hardware transactional memory barriers, to improve the performance of a sequential implementation of Dijkstra's algorithm. In this case, a helper thread does useful work rather than just prefetching. In our approach, we employ helper threads to speedup transactional applications rather than sequential applications.

High-level parallel programming. One approach to support parallelism is to extend existing programming languages with keywords from the parallel programming domain to spawn and synchronize threads and partition data. This alternative is well exploited in languages such as Cilk++ [20] and Charm++ [27]. Particularly, Cilk++ [20] has extensively employed *work stealing* in order to overcome scalability bottlenecks. However, both Charm++ and Cilk++ are designed for nontransactional parallel applications. In contrast, our approach is based on skeletal programming for software transactional memory applications. An extensive survey of recent parallel skeleton languages and libraries is presented in [28]. Although many parallel skeletons have been proposed, they are efficient only for regular data and task-parallel applications. In [15], Kulkarni et al. have identified that new programming abstractions are required to efficiently use speculative execution on a particular class of irregular applications that exhibits amorphous data-parallelism [29]. These applications are mostly worklist-based algorithms that work within a shared graph data structure. They have implemented the Galois system, a Java-based interface that provides a set of iterators to implement the worklist and a runtime system that implements speculative execution support.

9 Conclusions

Although many parallel programming models and systems have been proposed in the recent past, improving performance and programmability of applications is still hard and error prone. In this paper, we presented a new skeleton framework for transactional worklist applications. As presented, in addition to simplifying the programming task, it enables various performance optimizations transparently to the application programmer. Its novel autotuning mechanism automatically selects and adjusts optimizations to provide the best performance for each application. Results showed that our autotuned skeleton framework achieves similar or better performance than a static oracle.

As future work, OpenSkel can be extended to accommodate more patterns. For instance, the pipeline pattern can be created by composing a sequence of worklists. Additionally, the autotuning mechanism can be enhanced by considering new optimizations and STM configurations. As new TM worklists become available, it will be also possible to evaluate the autotuning mechanism using cross validation. Finally, we intend to integrate OpenSkel to the Dresden TM compiler and extend it to produce helper thread code.

Acknowledgments

This work was supported by the Scottish Informatics and Computer Science Alliance (SICSA), the Institute for Computing Systems Architecture (ICSA), and the University of Edinburgh. This is work that Dr. Góes performed while being a PhD student at the University of Edinburgh.

References

[1] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-Scale Computing Research Overview," TR, Intel, 2006.

[2] M.J. Irwin and J.P. Shen, "Revitalizing Computer Architecture Research," *Proc. Conf. Grand Research Challenges,* 2005.

[3] E.A. Lee, "The Problem with Threads," *Computer,* vol. 39, no. 5, pp. 33-42, May 2006.

[4] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick, "A View of the Parallel Computing Landscape," *Comm. ACM,* vol. 52, no. 10, pp. 56-67, 2009.

[5] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press & Pitman, 1989.

[6] M. McCool, "Structured Parallel Programming with Deterministic Patterns," *Proc. Second USENIX Conf. Hot Topics in Parallelism (HotPar),* pp. 25-30, 2010.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Symp. Operating System Design and Implementation (OSDI),* pp. 137-150, 2004.

[8] J. Larus and R. Rajwar, *Transactional Memory.* Morgan & Claypool Publishers, 2006.

[9] T. Karcher and V. Pankratius, "Run-Time Automatic Performance Tuning for Multicore Applications," *Euro-Par: Proc. 17th Int'l Conf. Parallel Processing,* pp. 3-14, 2011.

[10] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP),* pp. 237-246, 2008.

[11] R.M. Yoo and H.-H.S. Lee, "Adaptive Transaction Scheduling for Transactional Memory Systems," *Proc. 20th Ann. Symp. Parallelism in Algorithms and Architectures (SPAA),* pp. 169-178, 2008.

[12] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming.* Pearson Education, 2004.

[13] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* OREILLY, 2007.

[14] K. Fraser and T. Harris, "Concurrent Programming Without Locks," *ACM Trans. Computer Systems,* vol. 25, no. 2, article 5, 2007.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P.L. Chew, "Optimistic Parallelism Requires Abstractions," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* pp. 211-222, 2007.

[16] M.F. Spear, L. Dalessandro, V.J. Marathe, and M.L. Scott, "A Comprehensive Strategy for Contention Management in Software Transactional Memory," *Proc. 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP),* pp 32-40, 2009.

[17] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C.C. Kirkham, and I. Watson, "Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering," *Proc. Fourth Int'l Conf. High Performance Embedded Architectures and Compilers (HiPEAC),* pp. 4-18, 2009.

[18] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M.A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali, "Structure-Driven Optimizations for Amorphous Data-Parallel Programs," *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP),* pp. 3-14, 2010.

[19] Y. Song, S. Kalogeropulos, and P. Tirumalai, "Design and Implementation of a Compiler Framework for Helper Threading on Multicore Processors," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 99-109, 2005.

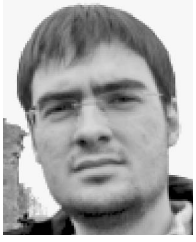
[20] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *J. Parallel and Distributed Computing,* vol. 37, no. 1, pp. 55-69, Aug. 1996.

[21] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," *Proc. 28th Ann. Int'l Symp. Computer architecture (ISCA),* pp. 14-25, 2001.

[22] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC),* pp. 35-46, 2008.

[23] W. Baek, C.C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun, "The Opentm Transactional Application Programming Interface," *Proc. 16th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 376-387, 2007.

[24] V.J. Marathe, W.N. Scherer III, and M.L. Scott, "Adaptive software Transactional Memory," *Proc. 19th Int'l Conf. Distributed Computing (DISC)*, pp. 354-368, 2005.
[25] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a Theory of Transactional Contention Managers," *Proc. 24th Ann. ACM Symp. Principles of Distributed Computing (PODC)*, pp. 258-264, 2005.
[26] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris, "Employing Transactional Memory and Helper Threads to Speedup Dijkstras Algorithm," *Proc. Int'l Conf. Parallel Processing (ICPP)*, pp. 388-395, 2009.
[27] L. Kale and S. Krishnan, "Charm++: A Portable Concurrent Object Oriented System Based on c++," *Proc. Eighth Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, pp. 91-108, 1993.
[28] H. González-Vélez and M. Leyton, "A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers," *Software Practice Experiments*, vol. 40, pp. 1135-1160, 2010.
[29] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval, "How Much Parallelism Is There in Irregular Applications?" *Proc. 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, pp. 1-12, 2009.



Luís Fabrício Wanderley Góes received the BSc degree in computer science from PUC Minas in 2002, the MSc degree in electrical engineering from PUC Minas in 2004, and the PhD degree from the University of Edinburgh in 2012. He is currently an associate professor at PUC Minas. His research interests include parallel programming patterns, software transactional memory and parallel job scheduling. He is a member of the IEEE.

Nikolas Ioannou received the Diploma in electrical and computer engineering from the National Technical University of Athens in 2008 and the PhD degree from the University of Edinburgh in 2012. He is currently a research assistant at the University of Edinburgh. His research interests include parallel computer architectures, many-core systems, thread-level speculation, and power management. He is a member of the IEEE and a student member of the ACM.

Polychronis Xekalakis received the Dipl Eng from the University of Patras in 2005 and the PhD degree from the University of Edinburgh in 2009. He is currently a senior research scientist at Intel-Labs Barcelona. His research interests include co-designed virtual machines, speculative multithreading, and architectural techniques for low power. He is a member of the IEEE.

Murray Cole is currently an associate professor of computer science at the University of Edinburgh. He is a member of the Institute for Computing Systems Architecture, within the School of Informatics. His research interests include parallel programming models, emphasizing approaches which exploit skeletons to package and optimize well-known patterns of computation and interaction as parallel programming abstractions. He is a senior member of the IEEE.

Marcelo Cintra received the BS and MS degrees from the University of Sao Paulo in 1992 and 1996, respectively, and the PhD degree from the University of Illinois at Urbana-Champaign in 2001. After completing the PhD degree, he joined the faculty of the University of Edinburgh, where he is currently an associate professor. His research interests include parallel architectures, optimizing compilers, and parallel programming. He has published extensively in these areas. He is a senior member of the ACM, the IEEE, and the IEEE Computer Society. More information about his current research activities can be found at http://www.homepages.inf.ed.ac.uk/mc.